ADAPTIVE INSTRUMENTATION RUNTIME MONITORING AND ANALYSIS

TECHNICAL FIELD

The present invention relates to program instrumentation for run-time software monitoring and analysis.

BACKGROUND

Static checking for program correctness, while currently an area of great promise and ongoing investigation, is fundamentally unable to detect large classes of program defects that are nevertheless very important. Software testing addresses these shortcomings but is expensive and limited by the scenarios used, and the amount of testers and time allotted. Consequently, software continues to ship with latent bugs, which can compromise system security in addition to affecting reliability. Furthermore, even "correct" software can violate a higher-level security policy. Given this, a pragmatic approach is to self-monitor application execution and report encountered defects.

One method for finding latent software defects is runtime monitoring. Runtime monitoring of an executing program can identify many program defects that static checking may miss, such as memory leaks, data races, and invariance. Runtime monitoring can be implemented by instrumenting code. Instrumentation refers to the process of adding code to software that monitors and collects data regarding memory management and other details of the software at runtime. However, currently, the overhead added to the executing program can exceed 30%, a slowdown that users are likely to notice and software developers are unlikely to accept. One solution is burst sampling.

The sequence of all events occurring during execution of a program is generally referred to as the "trace." A "burst" on the other hand is a subsequence of the trace.

Arnold and Ryder present a framework that samples bursts. (*See*, M. Arnold and B. Ryder, "A Framework For Reducing The Cost Of Instrumented Code," Programming Languages Design And Implementation (PLDI) (2001).) In their framework, the code of each procedure is duplicated. (*Id.*, at Figure 2.) Both versions of the code contain the original instructions, but only one version is instrumented to also collect profile information. The other version only contains checks at procedure entries and loop back-edges that decrement a counter "nCheck," which is initialized to "$nCheck_0$." Most of the time, the (non-instrumented) checking code is executed. Only when the nCheck counter reaches zero, a single intraprocedural acyclic path of the instrumented code is executed and nCheck is reset to $nCheck_0$.

A limitation of the Arnold-Ryder framework is that it stays in the instrumented code only for the time between two checks. Since it has checks at every procedure entry and loop back-edge, the framework captures a burst of only one acyclic intraprocedural path's worth of trace. In other words, only the burst between the procedure entry check and a next loop back-edge is captured. This limitation can fail to profile many longer "hot data stream" bursts, and thus fail to optimize such hot data streams. Consider for example the code fragment:

```
for (i=0; i<n; i++)
        if (...) f();
        else g();
```

Because the Arnold-Ryder framework ends burst profiling at loop back-edges, the framework would be unable to distinguish the traces fgfgfgfg and ffffgggg. For optimizing single-entry multiple-exit regions of programs, this profiling limitation may make the difference between executing optimized code most of the time or not.

Another limitation of the Arnold-Ryder framework is that the overhead of the framework can still be too high for dynamic optimization of machine executable code binaries. The Arnold-Ryder framework was implemented for a Java virtual machine execution environment, where the program is a set of Java class files. These Java programs typically have a higher execution overhead, so that the overhead of the instrumentation checks is smaller compared to a relatively slow executing program. The overhead of the Arnold-Ryder framework's instrumentation checks may make dynamic optimization with the framework impractical in other settings for programs with lower execution overhead (such as statically compiled machine code programs).

A framework that supports bursty tracing for low-overhead temporal profiling is described in Chilimbi, T. and Hirzel, M., "Bursty Tracing: A Framework for Low-Overhead Temporal Profiling," in *Workshop on Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001; and Chilimbi, T. and Hirzel, M., "Dynamic Hot Data Stream Prefetching For General-Purpose Programs," in *PLDI '02*, June 17-19, 2002. This bursty tracing framework adds a second counter (nInstr) that controls the length of execution in the instrumented version of the code. In this way, the bursty tracing framework can periodically capture complete program execution detail (i.e., a "trace sample") for short timeframes. Further, bursty tracing permits additional control and flexibility by allowing the "trace sample" extent to be configured in addition to collection frequency, by use of the two user-specified counters (nCheck and nInstr).

A drawback of the bursty tracing is that its sampling methodology may miss infrequently executed code paths that are nevertheless important for identifying reliability and/or security problems.

SUMMARY

A technique for bursty tracing with adaptive instrumentation provides improved low-overhead temporal profiling and analysis of software, with an emphasis on checking correctness of software rather that improving performance. The adaptive instrumentation addresses the shortcomings of the prior bursty tracing framework by adjusting the sampling rate of instrumented procedures according to their frequency of

execution. For example, the technique can sample execution of code paths at a rate inversely proportional to their execution frequency. In this way, rarely executed code paths can be essentially always traced, whereas frequently executed code paths are sampled at a very low rate.

In accordance with one implementation of the technique described herein, a bursty profiling framework is extended by also placing a counter at each of one or more dispatch check points to track the frequency of execution of the code path. The sampling rate at which the respective dispatch check point diverts execution to an instrumented version of the code is then varied according to the frequency of execution of the code path as tracked by this counter. For example, the software may begin execution with each of the dispatch check points providing a 100% sampling rate, which is then gradually adjusted downward according to the execution count of the code path.

In accordance with a further technique described herein, the bursty tracing adaptive instrumentation framework is applied to detecting memory leaks in software. In particular, the sampling of bursty trace data obtained by this instrumentation framework during runtime monitoring is analyzed to determine all objects that satisfy a staleness predicate as memory leaks.

Additional features and advantages of the invention will be made apparent from the following detailed description of embodiments that proceeds with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a program modified according to a bursty tracing framework.

Figure 2 is a framework for utilizing adaptive instrumentation in order to provide runtime monitoring and analysis.

Figure 3 is a block diagram of a memory leak detection tool implemented using the frame work of Figure 2.

Figure 4 shows a flowchart for one implementation of a memory leak detection tool.

Figure 5 is a block diagram of a data race detection tool implemented using the frame work of Figure 2.

Figure 6 illustrates a generalized example of a suitable computing environment 600 in which the described techniques can be implemented.

DETAILED DESCRIPTION

The following description is directed to techniques and tools for bursty tracing with adaptive instrumentation for low-overhead, temporal profiling, which can be used in runtime monitoring and analysis of software. The techniques and tools can use a bursty tracing with adaptive instrumentation framework structured to comprise duplicate non-instrumented ("checking code") and instrumented code versions of at least some original procedures of the program.

A bursty tracing framework is described briefly in reference to Figure 1. Figure 1 is a block diagram of a program modified according to a bursty tracing framework 100. Original procedures 110 in the target software are duplicated such that a checking code (original code) 120 is produced along with an instrumented code 130. The instrumented code 130 can contain any number of instrumentation points. The framework 100 can comprise dispatch checks 140-141, which can be placed, for example, at procedure entry and loop back-edges. The dispatch checks are responsible for diverting control between the checking and instrumentation copies of the code based on counters relating to sample size (nInstr) and sampling period or rate (nCheck). Further details of the bursty tracing framework are provided in Trishul A. Chilimbi and Martin Hirzel, "Dynamic Temporal Optimization Framework," U.S. Patent

Application No. 10/305,056, filed November 25, 2003, which is hereby incorporated herein fully by reference.

While a bursty tracing framework captures temporal execution detail of frequently executed code paths, many program defects only manifest on rarely visited code regions that periodic bursty trace sampling alone is likely to miss. This shortcoming can be overcome by a variation of the bursty tracing framework that includes adaptive instrumentation, where a sampling rate at which bursty traces of the instrumented code are sampled is adapted to the frequency of execution of the respective code path.

In this bursty tracing with adaptive instrumentation, the framework maintains a per-dispatch check sampling rate rather than a global sampling rate. More specifically, a separate set of nCheck and nInstr counters are associated with each adaptive dispatch check. As in the bursty tracing framework, the nCheck counter for the adaptive dispatch check counts down a number of executions of the dispatch check that dispatch execution into the checking code. So long as the nCheck counter is non-zero, the dispatch check dispatches to the checking code 120. When the nCheck counter reaches zero, the nInstr counter is initialized to its initial value $nInstr_0$, and execution is dispatched into the instrumented version of the code 130. The nInstr counter counts down a number of executions dispatched to the instrumented code 130. When the nInstr counter again reaches zero, the nCheck counter is re-initialized to its initial value $nCheck_0$. The initial values $nCheck_0$ and $nInstr_0$ then determine the period between bursty trace samples and the length of the bursty trace, respectively. The sampling rate (r) is then given by $r = nInstr_0/(nCheck_0 + nInstr_0)$.

In bursty tracing with adaptive instrumentation, this sampling rate is adapted to the frequency of execution of the code path through the adaptive dispatch check. The more often the code path (i.e., the adaptive dispatch check) is executed, the more the sampling rate is decreased. In one implementation, all adaptive dispatch checks initially produce bursty trace samples at a rate at or near 100% (full tracing). More specifically, the nCheck counter is initialized at $nCheck_0=0$, so that the first time the code path is executed a bursty trace is sampled through the code path. (As described above, if the nCheck counter is zero when the dispatch check is executed, then execution is dispatched to the instrumented code.) This provides a sampling rate of $r=nInstr_0/(0+nInstr_0)=1$ (which is 100% execution of the instrumented code).

On subsequent executions of the adaptive dispatch check, the counter nCheck that controls the sampling period is adapted to decrease the sampling rate. On subsequent resets of the nCheck counter, the sampling rate is decremented (by increasing the initialization value $nCheck_0$ of the nCheck counter at reset) towards a pre-set lower bound. In this fashion, rarely used code is sampled at very high rates, whereas more frequently executed code is sampled at or near a much lower sampling rate. In other words, the adaptive dispatch checks of rarely used code paths have high sample rates, whereas those of more frequently executed code are varied to lower sampling rates.

For adapting the sampling rates of the adaptive dispatch checks, the bursty tracing with adaptive instrumentation framework employs a further counter on the adaptive dispatch check and set of parameters, which includes a decrement, an interval, and a bound. This further counter controls when the sampling rate is decremented, and can be a count of the number of executions of the adaptive dispatch check in one implementation; or alternatively, a number of bursty traces sampled from the adaptive dispatch check (e.g., a number of nCheck counter resets), among other alternatives. The decrement determines how much to decrement the sampling rate (how much to increase the initialization value of the nCheck counter each time that it is reset). For example, in one implementation, the sampling rate is decremented by a factor of 10 each time the sampling rate is decreased, e.g., from 100%, 10%, 1%, 0.1%, etc. The interval determines how often to decrement the sampling rate. In one implementation, the sampling rate is decremented progressively less often. For example, the interval between decrements can be increased by a factor of 10 each time time the sampling rate is decremented, e.g., from an interval of 10 nCheck counter resets, to 100, 1000, 10,000, etc. The bound counter determines the lower bound of the sampling rate for the adaptive dispatch check.

In one implementation, adaptation of the sampling rate is then performed when the nCheck counter is reset. In one implementation beginning with a 100% sampling rate, nCheck is initially set to zero, resulting in dispatching to the instrumented code to sample a bursty trace. On a subsequent execution of the adaptive dispatch check after the bursty trace completes, the nCheck counter is reset, after first adapting (possibly decrementing) the sampling rate.

In adapting the sampling rate, the interval counter determines the interval at which the sampling rate is decremented. The interval counter is incremented at each nCheck counter reset, and causes a decrement each time its count reaches a pre-set limit. This interval progressively increases. For example, in one implementation, the interval limit is increased by a factor of 10 each time it is reached (i.e., the interval limit is increased by interval limit=10*interval limit, each time it is reached), so that decrements are performed at nCheck reset counts of 10, 100, 1000, 10,000, etc.

At each decrement of the sampling rate, the initialization value ($nCheck_0$) of the nCheck counter is increased so as to effect the decrement in the sampling rate (r). In one implementation, the value of nCheck is varied according to the formula $nCheck_0(n) = (decr^{n-1}-1)*nInstr0$, so $nCheck_0(1) = 0$, $nCheck_0(2) = 9*nInstr_0$, $nCheck_0(3) = 99*nInstr_0$. With decr=10, then this formula yields: r(1) = 100%, r(2) = 10%, r(3) = 1%. The decrement in the sampling rate continues until the lower bound of the sampling rate (e.g., 0.1% in one implementation) is reached.

For instance, in one embodiment, all dispatch checks can be sampled at a rate of 100% (i.e., full tracing) initially. Subsequent executions of the adaptive dispatch check then progressively reduce the sampling rate by an adjustable fractional amount until the adjustable or user-set lower bound sampling rate is reached. In the steady state, rarely executed code segments are virtually traced (sampled at close to 100%), while frequently executed code paths are sampled at the lower bound sampling rate. This approach trades the ability to distinguish frequently executed code paths from infrequently executed ones for more comprehensive code coverage.

The parameters of the adaptive dispatch checks can be adjusted in alternative implementations of the framework, so as to provide a different starting sampling rate, a different decrement size or factor, different interval of decrement progression, and a different lower bound, among others.

Adaptive instrumentation can be used as a monitoring technique for plug-in tools that use the information to analyze the "correctness" of software, including but not limited to data races, memory leaks, and invariance. A framework for utilizing adaptive instrumentation in order to provide runtime monitoring and analysis is illustrated in Figure 2. In one embodiment, infrequent program events, such as dynamic heap allocations and lock acquisitions, can be traced using conventional instrumentation. Such instrumentation 202 can therefore be placed directly into the checking code 204, as shown in Figure 2, to provide runtime data on those events (Record Rare Event()).

Frequent events that are too expensive to trace, such as data references, branch executions, memory allocations, synchronization events (locks, barriers, critical sections, semaphores), load and stores, branches, etc., can be monitored using adaptive instrumentation or bursty tracing. These events can be monitored by adding instrumentation points 208 to the instrumented code 206 (Record Freq Event 1/2 ()). The instrumentation points 208 can be added at custom locations in the instrumented code or at each of the previously mentioned events. Once instrumentation points 208 are provided, plug-in code can simply be provided at the instrumentation points that are executed at each of the monitored event occurrences to provide runtime data to an analysis tool 210.

In one embodiment, a memory leak detection tool is provided that uses the disclosed monitoring techniques to detect memory leaks if a heap object has not been accessed for a "long" time. This simple invariant ensures that the tool detects all leaks that manifest at runtime. However, there are two significant obstacles to implementing this staleness policy within a practical memory leak tool. First, the overhead of monitoring all heap data accesses can be prohibitive. Second, the leaks reported could include a large number of "false positives". Perhaps for these reasons, no existing memory leak tool uses this "staleness" approach.

The tool of the current embodiment addresses the first problem by using adaptive instrumentation or bursty tracing techniques to monitor heap accesses with low overhead. For instance, in this embodiment, the tool can use a lower bound sampling rate of 0.1%, which may entail a runtime overhead of less than 5% in some instances. Regarding "false positives," tuning the "time elapsed" before an unaccessed heap object is reported as a leak is sufficient to remedy this problem. In addition, many of the remaining false positives are of interest to developers since objects that have not been accessed for a very long time often indicate inefficient use of memory. Sampling the heap data accesses appears to have no noticeable impact on the number of false positives. This may be so because most heap objects are accessed multiple times and the sampling will have to miss all of these accesses for an active object to be mistakenly classified as a leak.

Figure 3 is a block diagram of a memory leak detection tool implemented using the framework of Figure 2. Figure 3 shows checking code 302 and instrumented code 304. Checking code 302 includes instrumentation code 306 for providing information regarding memory heap allocations and frees to memory leak detection tool 308. Instrumentation code 306 includes instrumentation points where code can be inserted in order to provide event information. In this embodiment, the instrumentation points contain code 310 that provides sampled sets of heap accesses to memory leak detection tool 308.

Figure 4 shows a flowchart for one implementation of a memory leak detection tool. An instrumented version of the software 400 is created at block 402. This instrumented version can be created using the adaptive instrumentation and bursty tracing techniques described previously. Additionally, mapping information 404, such as mapping a CPU instruction counter value at runtime to a line in source code, is also provided to facilitate "last access" information. The instrumented version is executed at block 406 in place of the original code along with a memory leak detection tool 408. The instrumented application can communicate event information, such as heap allocations and frees via instrumentation in the checked code, while a sampled set of heap accesses obtained via runtime monitoring using adaptive instrumentation or bursty tracing is provided to the memory leak detection tool 408 as previously described in Figure 2.

The memory leak detection tool 408 uses the heap allocation and free information to maintain a model of the heap, which it updates with the heap access information. Periodically, the memory leak detection tool 408 takes a snapshot at block 410, where it visits all objects in its heap models and reports all objects that satisfy its staleness predicate as leaks. It is able to associate the responsible heap allocation, all heap frees that deallocated objects created at that allocation site, and most importantly the "last access", with each heap object reported as a leak. This last access information is invaluable for quickly debugging and fixing detected leaks. In addition, the last access information enables quick determination of "false positives". The leak snapshots are post processed at block 412 and leak reports can then be visualized through a user terminal at block 414. The user terminal can also include a source code browser that highlights a line of code in the program that is the last access to a leaked object.

A data race detection tool can also be implemented using the framework of Figure 2. A data race occurs when an object is accessed by two different threads without explicit synchronization, which can cause unexpected behavior typically not evidenced until quite some time down-stream in the execution path. A data race detection tool works by tracking lock acquisitions and releases and refining its model of which locks protect which shared data by keeping track of the locks held by threads that access shared variables. Each shared variable has a lock-set that is constantly refined by intersecting it with the locks held by the thread making the current access. If the intersection is null, the tool reports a race. The tool must incur a high runtime overhead because it has to track all data accesses. By implementing a tool using the framework of Figure 2, this overhead can be significantly reduced.

Figure 5 is a block diagram of a data race detection tool implemented using the frame work of Figure 2. Figure 5 shows checking code 502 and instrumented code 504. Checking code 502 includes instrumentation code 506 for providing information regarding lock acquisitions and releases to data race detection tool 508. Instrumentation code 506 includes instrumentation points where code can be inserted in order to provide event information. In this embodiment, the instrumentation points contain code 510 that provides sampled sets of heap accesses to data race detection tool 508.

Figure 6 illustrates a generalized example of a suitable computing environment 600 in which the described techniques can be implemented. The computing environment 600 is not intended to suggest any limitation as to scope of use or functionality of the invention, as the present invention may be implemented in diverse general-purpose or special-purpose computing environments.

With reference to Figure 6, the computing environment 600 includes at least one processing unit 610 and memory 620. In Figure 6, this most basic configuration 630 is included within a dashed line. The processing unit 610 executes computer-executable instructions and may be a real or a virtual processor. In a multi-processing system, multiple processing units execute computer-executable instructions to increase processing power. The memory 620 may be volatile memory (e.g., registers, cache, RAM), non-volatile memory (e.g., ROM, EEPROM, flash memory, etc.), or some combination of the two. The memory 620 stores software 680 implementing the software described herein.

A computing environment may have additional features. For example, the computing environment 600 includes storage 640, one or more input devices 650, one or more output devices 660, and one or more communication connections 670. An interconnection mechanism (not shown) such as a bus, controller, or network interconnects the components of the computing environment 600. Typically, operating system software (not shown) provides an operating environment for other software executing in the computing environment 600, and coordinates activities of the components of the computing environment 600.

The storage 640 may be removable or non-removable, and includes magnetic disks, magnetic tapes or cassettes, CD-ROMs, CD-RWs, DVDs, or any other medium which can be used to store information and which can be accessed within the computing environment 600. The storage 640 stores instructions for adaptive instrumentation runtime monitoring and analysis software 680.

The input device(s) 650 may be a touch input device such as a keyboard, mouse, pen, or trackball, a voice input device, a scanning device, or another device that provides input to the computing environment 600. For audio, the input device(s) 650 may be a sound card or similar device that accepts audio input in analog or digital form, or a CD-ROM reader that provides audio samples to the computing environment. The output device(s) 660 may be a display, printer, speaker, CD-writer, or another device that provides output from the computing environment 600.

The communication connection(s) 670 enable communication over a communication medium to another computing entity. The communication medium conveys information such as computer-executable instructions, audio/video or other media information, or other data in a modulated data signal. A modulated data signal is a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media include wired or wireless techniques implemented with an electrical, optical, RF, infrared, acoustic, or other carrier.

The adaptive instrumentation runtime monitoring and analysis techniques herein can be described in the general context of computer-readable media. Computer-readable media are any available media that can be accessed within a computing environment. By way of example, and not limitation, with the computing environment 600, computer-readable media include memory 620, storage 640, communication media, and combinations of any of the above.

The techniques herein can be described in the general context of computer-executable instructions, such as those included in program modules, being executed in a computing environment on a target real or virtual processor. Generally, program modules include routines, programs, libraries, objects, classes, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The functionality of the program modules may be combined or split between program modules as desired in various embodiments. Computer-executable instructions for program modules may be executed within a local or distributed computing environment.

For the sake of presentation, the detailed description uses terms like "determine," "generate," "adjust," and "apply" to describe computer operations in a computing environment. These terms are high-level abstractions for operations performed by a computer, and should not be confused with acts performed by a human being. The actual computer operations corresponding to these terms vary depending on implementation.

In view of the many possible embodiments to which the principles of our invention may be applied, we claim as our invention all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.

Although the specific embodiments described herein referred to detection of memory leaks and data races, the invention is not so limited. For instance, a tool can be created using the disclosed techniques for checking invariance in code by simply altering the disclosed method to check array accesses versus array bounds in the instrumented versions of the original code.